El lenguaje Python

David Masip Rodó

PID_00174138



Índice

Introducción

En este módulo aprenderemos las bases del lenguaje de programación Python. Este módulo no pretende ser una guía exhaustiva de todas las características que este lenguaje de programación nos ofrece, sino que está enfocado a introducir al estudiante en el entorno Python. El objetivo del texto será ayudar a comprender los ejemplos que se exponen en los módulos de teoría, y proporcionar una base que permita al estudiante elaborar las distintas actividades que el curso requiere.

A nivel de pre-requisitos, no asumiremos ningún conocimiento previo de Python. Sin embargo, es muy importante tener conocimientos de algún lenguaje de programación previo (C, C++, JAVA, ...), puesto que los conceptos elementales (variable, bucle, función,...) no se tratarán en profundidad en este módulo.

El lenguaje Python fue diseñado a finales de la década de los ochenta por Guido van Rossum. Se trata de un lenguaje de programación de muy alto nivel, con una sintaxis muy clara y una apuesta firme por la legibilidad del código. Sin duda es un lenguaje de programación muy versátil, fuertemente tipado, imperativo y orientado a objetos, aunque contiene también características que lo convierten en un lenguaje de paradigma funcional.

Python se puede considerar un lenguaje semi-interpretado. A diferencia de C, el código Python no se ejecuta directamente en la máquina destino, sino que es ejecutado por un SW intermedio (o intérprete). Sin embargo, al igual que JAVA, Python compila el código escrito en lenguaje de alto nivel para obtener un pseudo código máquina (*bytecode*) que es el que propiamente ejecuta el intérprete. Existen versiones del intérprete de Python para la mayor parte de las plataformas.

En este módulo, explicaremos los pasos necesarios para instalar Python en una determinada plataforma. A continuación veremos las principales características del lenguaje a nivel sintáctico.

Objetivos

Los objetivos que el alumno debe de haber conseguido una vez estudiados los contenidos de este módulo son los siguientes:

1. Refrescar los conceptos elementales de programación aprendidos en cursos anteriores.
2. Aprender las bases del lenguaje Python.
3. Ser capaz de entender el código que se muestra en los ejemplos de teoría y poder modificarlo para realizar otras funcionalidades.
4. Usar características avanzadas de Python que permitan utilizarlo como lenguaje rápido para el diseño y codificación de prototipos de inteligencia artificial.

1. Instalación de Python

Existen actualmente múltiples implementaciones del lenguaje Python, en función del lenguaje base en el que se ha construido el intérprete. La más conocida es CPython, o simplemente Python, que ha sido implementada en lenguaje C. Otras opciones disponibles son: IronPython (codificada en C#) o JPython (codificada en JAVA).

Para instalar una determinada versión de Python, aconsejamos recurrir a su sitio web oficial*. En el momento de escribir estos manuales, esta URL contiene toda la información necesaria para instalar el entorno. En la sección de descargas** se pueden obtener los últimos paquetes de instalación para Windows, Linux y MacOS X en sus distintas versiones.

* http://www.python.org/

** http://www.python.org /download/

Una vez obtenido el paquete para el sistema operativo y versión adecuados, ejecutaremos el instalador, y en una línea de comandos ejecutaremos el intérprete de Python, y probaremos que todo funciona correctamente mediante el clásico "Hello World".

```
macbook:~ david$ python
Python 2.6.6 (r266:84292, Oct 16 2010, 21:41:03)
[GCC 4.0.1 (Apple Inc. build 5490)] on darwin
Type ‘‘help’’, ‘‘copyright’’, ‘‘credits’’ or ‘‘license’’ for
more information.
>>> print ‘‘Hello World’’
Hello World
>>> quit()
macbook:~ david$
```

En este ejemplo, hemos llamado al intérprete de Python desde el símbolo de sistema. Al iniciarse, después de imprimir la información de versiones, Python nos muestra el símbolo ">>>" para indicarnos que el intérprete está esperando comandos. En este pequeño ejemplo le pedimos que nos imprima la cadena de caracteres "Hello World", y después de ver el resultado, salimos de Python mediante la instrucción `quit()`, volviendo al símbolo de sistema. Llegados a este punto, podemos considerar que tenemos la instalación básica de Python lista para su uso.

En este módulo trabajaremos fundamentalmente con un editor de textos, y ejecutaremos los scripts, que programaremos mediante el intérprete. En el ejemplo anterior, podríamos haber guardado la línea `print "Hello World"` en un archivo llamado Hola.py, y ejecutarlo mediante el intérprete con la instrucción:

```
macbook:~ david$ python Hola.py
Hello World
```

Entornos de desarrollo IDE

Existe la posibilidad de instalar entornos de desarrollo IDE específicos para Python. No es el objetivo de este módulo analizar los IDE disponibles en el mercado actualmente, aunque citaremos los tres entornos libres más conocidos. Por un lado PyDEV* es un conocido *plug in* de Python para el entorno de desarrollo Eclipse, bastante utilizado en el sistema universitario. También existe la opción de usar un entorno propio, como SPE (Stani's Python Editor)** , o Wing***. En este módulo trabajaremos directamente contra el intérprete de comandos, cualquier editor de textos simple será suficiente para las funcionalidades que desarrollaremos.

* http://pydev.org/

** http://sourceforge.net/projects/spe/

*** http://www.wingware.com/

2. Variables

Los tipos básicos del lenguaje Python son esencialmente los ya conocidos en cualquier lenguaje de programación: los valores numéricos, las cadenas de texto, y los valores booleanos. En el código 2.1 se muestra un ejemplo de cada tipo.

Código 2.1: Ejemplos de uso de variables

```
# Ejemplos de variables

a =42;    #valor entero
along = 42L; #valor entero long (32–64 bits en función de la plataforma)
ahex = 0x2a;  #a en en notación hexadecimal
aoctal = 052;  #a en notación octal

b = 3.1416;  #valor en coma flotante
bnotacion = 3.14e0;
c = 3 + 7j;  #Python soporta números complejos
d = "Ejemplo de cadena de caracteres" #Una cadena de caracteres

#Imprimir las variables por pantalla
print a,b,c,d;

#tipo de las variables
type(a);
type(b);
type(c);
type(d);
```

Los números enteros se pueden representar en notación decimal, octal (anteponiendo un "0" al valor) o hexadecimal (anteponiendo "0x" al valor). Al igual que en lenguaje C (en el que está escrito Python), los números se pueden representar mediante enteros (por defecto) o *long* (entero largo), que permite un rango mayor de valores posibles. Este rango, de nuevo como en C, dependerá de la plataforma subyacente, pudiendo ser de 32 o 64 bits.

Los valores flotantes son implementados a bajo nivel directamente con el tipo double de C (registro de 64 bits). Por su parte, los números complejos están soportados de base y en la práctica se implementan mediante dos flotantes, uno para la parte real y otro para la parte imaginaria.

2.1. Operadores

Los operadores que soporta el lenguaje Python son los clásicos de cualquier lenguaje de programación. Las tablas 1 y 2 muestran un resumen con ejemplos de estas operaciones. Además, Python provee de extensiones para operaciones más complejas en el módulo *math*.

Taula 1. Operadores con los tipos básicos

Nombre	Suma	Resta o negación	Multiplicación	División	Módulo	División entera	Exponenciación
Operador	+	-	*	/	%	//	**
Ejemplo	$a = 1 + 2$ #$a = 3$	$a = 2 - 5$ o $a = -3$ #$a = -3$	$a = 2 * 3$ #$a = 6$	$a = 5{,}4/2$ #$a = 2{,}7$	$a = 5\%2$ #$a = 1$	$a = 5{,}8//2$ #$a = 2{,}0$	$a = 3 * *4$ #$a = 81$

Taula 2. Operadores a nivel de bit

Nombre	And	Or	Xor	Not	Desplazamiento a la derecha	Desplazamiento a la izquierda
Operador	&	$\vert$	$\wedge$	$\sim$	$\gg$	$\ll$
Ejemplo	$a = 2\&1$ #$a = 0(10\&01)$	$a = 2\vert 1$ #$a = 3(10\vert 01)$	$a = 3 \wedge 1$ #$a = 1$	$a =\sim 1$ #$a = -2$	$a = 4 \gg 2$ #$a = 1$	$a = 1 \ll 2$ #$a = 4$

El valor de retorno de un operador en Python viene determinado por el tipo de las variables que intervienen en la operación. Así, por ejemplo, si sumamos dos números enteros, nos devolverá otro valor entero. Si uno de los dos operandos es un valor en punto flotante, el resultado de la operación será también un valor en punto flotante.

El tipo básico booleano puede recibir dos valores (*true, false*), y se utilizan fundamentalmente para expresar el resultado de condiciones, especialmente útiles en los bucles y control de flujo condicional. En la tabla 3 se resumen los distintos operadores que trabajan con valores booleanos, comunes a la mayor parte de lenguajes de programación.

Taula 3. Operadores con booleanos

Operador	And	Or	Not	==	! =	$<$	$>$
Ejemplo	*b = False and True*	*b = False or True*	*b = Not False*	8 == 9	8! = 9	$8 < 9$	$8 > 9$
Resultado	b será False	b será True	a=n será True	False	True	True	False

2.2. Cadenas de caracteres

Las cadenas de caracteres son fragmentos de texto delimitados en Python por comillas simples ('Ejemplo de cadena') o dobles ("Ejemplo de cadena"). Si queremos introducir saltos de línea, disponemos del carácter de escape '\ n'. También podemos usar otros caracteres de escape tradicionales como '\t' (tabulación) o '\b' (borrar carácter). Para conseguir que el texto se imprima tal y como aparece en el código fuente sin tener que recurrir a los códigos de escape, podemos delimitarlo por triples comillas (podéis ver un ejemplo en el código 2.2).

Código 2.2: Ejemplos de uso cadenas de caracteres

```
# Ejemplos de cadenas

a = "hola\n"
b = "\t␣Este␣es␣un␣ejempp\blo␣de␣cadenas␣de␣caracteres\n"
c = """Es␣posible␣escribir␣saltos␣de␣linea
sin␣necesidad␣de␣códigos␣de␣escape."""

print a,b,c

>>>hola
```

```
        Este es un ejemplo de cadenas de caracteres
Es posible escribir saltos de linea
sin necesidad de códigos de escape.

d = a + b;  #concatenación
e = "repite"
f = 3*e;    #repetición
g = e*3;    #equivalente al caso anterior

print d,f,g
```

Algunos operadores usados con valores numéricos se encuentran sobrecargados en el caso de las cadenas de caracteres. Son ejemplos de ello la igualdad (que asigna una cadena a una variable) y el operador suma "+". La suma de dos cadenas es el resultado de concatenar la segunda detrás de la primera. Del mismo modo, el producto de una cadena por un escalar da por resultado la misma cadena repetida tantas veces como indique el operador numérico.

Por último, una función muy útil en Python es **str**, que permite realizar la conversión de valores numéricos a cadena de caracteres.

Ejemplo

Si tecleamos *str(8.987)* en la línea de comandos, obtendremos la salida *'8.987'*.

3. Control de flujo

Todos los lenguajes de programación ponen a nuestra disposición instrucciones de control de flujo. Estas instrucciones permiten al programador alterar el orden secuencial del código con el fin de permitir ejecutar diferentes órdenes en función de una serie de condiciones sobre el estado. Python ofrece los dos tipos básicos de sentencias de control de flujo: las **sentencias condicionales** y los **bucles** (o repeticiones).

3.1. Sentencias condicionales: la instrucción if

La instrucción condicional **if** (del inglés *si*) recibe como entrada una expresión booleana, y sirve para ejecutar una de porción de código en función de si se cumple esta condición (el resultado de su evaluación es *true*).

La sintaxis de la instrucción consiste en la palabra clave **if**, a continuación la expresión booleana de la condición, y un signo **:** que indica el final de la condición, y finalmente, el código a ejecutar en caso de que la condición se evalúe a *true*.

En el código 3.1 tenéis un ejemplo de instrucción **if**.

Código 3.1: Ejemplo de sintaxis de un bloque if

```
# Ejemplos de uso de if

c = 37;
if c > 0:
        print "El número es positivo\n"
        print "Que tenga un buen día\n"
# continuación del programa
```

Un detalle muy importante a observar en este ejemplo de código es la **identación**. En Python los bloques de código se delimitan mediante una correcta tabulación. A diferencia de otros lenguajes donde disponemos de palabras clave específicas (*begin*, *end*,...) o llaves ({,..., }) reservadas para definir bloques de código, en Python sólo podemos usar tabulación.

La identación es un rasgo muy característico del código Python, y permite entre otras cosas una lectura mucho más agradable de los programas, y una fácil identificación de las distintas partes.

3.1.1. Sentencias if-elif-else

Cuando el objetivo de la sentencia condicional es dividir la ejecución del código en función de si se cumple la condición o no, y queremos que se haga alguna acción de forma explícita cuando la condición no se cumple, usamos la palabra clave **else** (siempre seguida de **:**).

Existe una tercera palabra clave en las instrucciones de control de flujo condicionales, **elif**, que se utiliza para añadir más condiciones a la sentencia if. Esto nos puede ser muy útil si tenemos muchos casos que diferenciar y queremos tratarlos todos. Se pueden añadir tantos bloques elif como queramos. En el código 3.2 podemos ver un ejemplo de uso del entorno if-elif-else.

Código 3.2: Ejemplo de sintaxis de un bloque else

```
# Ejemplos de uso de if

c = -10;
if c > 0:
        print "El número es positivo\n"
        print "Que tenga un buen día\n"
elif c == 0:
        print "El número es exactamente 0"
else:
        print "El número es negativo"
        print "Que tenga suerte"
# continuación del programa
```

Si el valor de c fuera 0, se ejecutaría la línea print "El número es exactamente 0". Cuando no se cumple la condición del if, ni de ninguno de los elif que pudiera haber, se acaba ejecutando el bloque de instrucciones correspondientes al **else**.

Finalmente, existe una forma compacta de representar sentencias condicionales (de forma similar a como usamos *Cond ? acción si A es cierto: acción si Cond es falso* en C) en Python. Se trata de los bloques **Accion1 if Cond else Accion2**. En este caso, se comprueba la condición *Cond* y se ejecuta *Accion1* si es *true* o *Accion2* si es *false*.

3.2. Secuencias iterativas: bucles

Los bucles son estructuras de control de flujo que permiten repetir un bloque de código un determinado o indeterminado número de veces. Esencialmente existen dos tipos de bucles: **for** y **while**.

3.2.1. Bucles for ... in

El bucle **for...in** se utiliza para ejecutar una secuencia de pasos un determinado número de veces (ya conocido previamente).

> La sintaxis que se utiliza es: **for** elemento **in** secuencia: y a continuación el bloque de acciones con la identación correcta.

En el ejemplo del código 3.3 podemos ver un uso típico del bucle for para recorrer un vector de datos.

Código 3.3: Ejemplo de sintaxis de un bucle for...in

```
# Ejemplos de uso del bucle for

vector = ["hola", "buenos", "días"];
for palabra in vector:
        print palabra
# imprimiría todas las palabras del vector
```

> Es importante destacar que el bucle for no se comporta exactamente como estamos acostumbrados en la mayor parte de lenguajes de programación. En Python, a cada paso del bucle se instancia el elemento iterador con un valor de la secuencia de forma automática.

En el ejemplo, la variable *palabra* irá cogiendo a cada paso el valor de uno de los elementos de la lista *vector* (más adelante veremos más detalles sobre el funcionamiento de las listas): en la primera iteración *palabra* tomará el valor “hola”, en la segunda valdrá automáticamente “buenos”, y así sucesívamente. No será necesario trabajar con índices puesto que Python hará todo el trabajo por nosotros.

3.2.2. while

El bucle while sigue un planteamiento mucho más similar a los lenguajes de programación tradicionales. Este bucle nos permite ejecutar un determinado bloque de código mientras una determinada condición sea cierta.

> Su sintaxis es: **while** Condición:

En el ejemplo del código 3.4 se muestra un uso típico del bucle while.

Código 3.4: Ejemplo de sintaxis de un bucle while

```
# Ejemplos de uso del bucle for

pares = 2;
while pares <= 20:
        print pares
        pares = pares + 2
# imprimiría todos los números pares
```

El ejemplo imprime por pantalla todos los números pares del 2 al 20. A cada paso incrementa la variable de control que nos permitirá salir del bucle cuando la condición no sea satisfecha.

Notad que la sentencia while puede generar bucles infinitos. A veces estos bucles se generan de forma involuntaria, lo cual va en detrimiento de la calidad de nuestros programas. Otras veces se generan de forma voluntaria, en combinación con la sentencia **break**. Esta sentencia permite salir de forma incondicional de un bucle.

El código 3.5 es equivalente al anterior, aunque mucho menos elegante. La sentencia **break** sin embargo puede ser útil cuando el flujo de ejecución depende de alguna entrada del usuario no contemplada en el momento de ejecución.

Código 3.5: Ejemplo de sintaxis de un bucle while

```
# Ejemplos de uso del bucle for

pares = 2;
while pares > 0:
        print pares
        pares = pares + 2
        if pares > 20:
                break
# imprimiría todos los números pares
```

4. Funciones

Todos los lenguajes de programación ofrecen mecanismos que permiten encapsular una serie de operaciones de forma parametrizable, y devolver unos resultados determinados. Estos fragmentos de código se denominan funciones.

> En Python las funciones se declaran mediante la palabra clave **def**. Su sintaxis es:
>
> **def** nombre_funcion(parametro1, parametro2, ..., parametroN):

A continuación se escribe un salto de línea y las operaciones asociadas a la función, como siempre con una identación a la derecha (tabulación) que indica que este código se corresponde a la función.

Delimitación de código fuente

Hay que recordar que Python no usa llaves ni palabras clave para delimitar código fuente.

Una vez se ha definido la función, se puede llamar con distintos parámetros dentro del código.

> La llamada o ejecución de una función sigue la sintaxis: nombre_función(parametro1, parametro2,..., parametroN).

4.1. Parámetros de entrada

Las llamadas a funciones reciben como argumentos una lista de parámetros (implementada en Python como una tupla), que se corresponden con los parámetros establecidos en el momento de la definición de la función. En el ejemplo del código 4.1 se puede observar la llamada a una función que imprime el resultado de elevar un número base (*parametro1*) a un exponente (*parametro2*).

Código 4.1: Ejemplo de sintaxis de una función

```
# Ejemplos de definiciones y llamadas de funciones
# Función potencia de dos valores (usa como base el primer parámetro
# y como exponente el segundo parámetro)

def exponenciar(parametro1, parametro2):
```

```
        ""Ejemplo de función que eleva un número o base a un exponente
        print parametro1**parametro2

def exponenciar2(parametro1, parametro2=2):
        print parametro1**parametro2

exponenciar(2,3);
exponenciar(parametro2=3,parametro1=2);
exponenciar(3,2);
exponenciar(parametro2=2,parametro1=3);

exponenciar2(5);
exponenciar2(5,3);

#llamadas erróneas
exponenciar(2,3,4);
```

En primer lugar notad que la tabulación vuelve a jugar un papel clave. Para diferenciar el final de la definición de la función y el resto de código del programa principal, basta con mirar el código fuente por encima. La propia identación delimita claramente los bloques.

También se puede observar una cadena inicial que empieza por “ ”. Estas cadenas se utilizan para documentar las funciones. Los programadores familiarizados con JAVA pueden ver cierto paralelismo con el doc usado en este lenguaje.

Al final de la definición de la función se encuentran las llamadas. Es interesante observar las distintas formas de cómo se pueden parametrizar las funciones en Python. El uso normal suele ser llamar a la función con los parámetros en el mismo orden en que se ha definido la función. Sin embargo, se puede también llamar a la función con el nombre del parámetro y su valor asociado. De este modo son posibles llamadas que se saltan la regla del orden como las del ejemplo, donde encontramos *exponenciar(parametro2=3, parametro1=2)*, que es equivalente a *exponenciar(2,3)*. Es importante destacar que Python obliga a llamar a las funciones usando exactamente el mismo número de argumentos, en caso contrario se generará un error de ejecución.

Existe también la posibilidad de inicializar parámetros con valores por defecto en caso de que no se proporcione su valor en el momento de la llamada. La función *exponenciar2* muestra un ejemplo donde en caso de no proporcionar un argumento utilizaría un 2 por defecto (elevaría al cuadrado).

Python permite también funciones con un número variable de parámetros. En el ejemplo del código 4.2 se puede ver la sintaxis utilizada en una función que imprime la suma de todos sus argumentos.

Código 4.2: Ejemplo de sintaxis de una función con parámetros variables

```
# Ejemplos de definiciones y llamadas de funciones
#función suma

def sumarLista( *parametros):
    resultado = 0;
    for val in parametros:
```

```
        resultado = resultado + val;
    print resultado;

sumarLista(1,2,3,4);
```

La sintaxis de este ejemplo es un poco confusa, puesto que aún no hemos tratado los tipos de datos complejos, como las listas y las tuplas. Esencialmente se define un argumento que contiene una tupla con todos los posibles parámetros a recibir. Esta tupla se recorre en tiempo de ejecución para poder utilizar los parámetros de forma adecuada.

Ved también

Las listas y las tuplas se estudian en el apartado 5 de este módulo.

Un último factor a tener en cuenta en los parámetros a funciones en cualquier lenguaje de programación es su modificación en tiempo de ejecución. Tradicionalmente los pasos por parámetro funcionan **por valor** o **por referencia**. En el primer caso los argumentos a una función no se modifican al salir de ella (en caso de que el código interno los altere). Técnicamente, se debe al hecho de que realmente no pasamos a la función la variable en cuestión, sino una copia local a la función que es eliminada al acabar su ejecución. En el caso del paso por referencia se pasa a la función un puntero al objeto (en el caso de C) o simplemente una referencia (en lenguajes de alto nivel) que permite su indirección. De este modo, las modificaciones a los parámetros que se hacen dentro de la función se ven reflejadas en el exterior una vez la función termina. En Python los pasos por parámetros son en general por referencia. La excepción la conforman los tipos de datos básicos o inmutables (enteros, flotantes, ...), que se pasan por valor. El código 4.3 muestra un ejemplo de este hecho.

Código 4.3: Ejemplo de paso por valor y referencia en Python

```
# Ejemplos de definiciones y llamadas de funciones
#función para verificar la modificabilidad de los parametros

def persistenciaParametros(parametro1, parametro2):
    parametro1 = parametro1 + 5;
    parametro2[1] = 6;
    print parametro1;
    print parametro2;

numeros = [1,2,3,4];
valorInmutable = 2;
persistenciaParametros(valorInmutable,numeros);
print numeros;
print valorInmutable;
```

Como se puede observar a partir de la ejecución del ejemplo, el vector de números sufre las modificaciones debidas al código interno de la función, mientras que el valor entero se modifica dentro de la función pero pierde esta modificación al devolver el control al programa principal.

4.2. Valores de retorno

Hasta ahora se han descrito las funciones Python como fragmentos de código que efectúan unas determinadas acciones de forma reutilizable. Los ejemplos

vistos son en el fondo versiones procedimentales de funciones, puesto que en ningún caso se devuelve un valor de retorno.

> La sintaxis para el retorno de valores en Python difiere de los lenguajes de programación habituales, puesto que Python permite devolver más de un valor de retorno. La palabra clave a utilizar es **return**, seguida de la lista de argumentos que hay que devolver.

Existe también la posibilidad de obviar los valores de retorno y utilizar el paso de parámetros por referencia, obteniendo un resultado similar. El código 4.4 muestra tres ejemplos de uso de retorno de valores en funciones. Notad que la función *sumaPotencia* devuelve el resultado de efectuar las dos operaciones en forma de lista. En realidad la función sólo devuelve un valor (la lista), pero el efecto producido es el mismo que si pudiera devolver múltiples valores.

Código 4.4: Ejemplo de retorno de valores en Python

```
# Ejemplos de definiciones y llamadas de funciones
#función para practicar el retorno de valores

def suma(parametro1, parametro2):
    return parametro1 + parametro2;

def potencia(parametro1, parametro2):
    return parametro1 ** parametro2;

def sumaPotencia(parametro1, parametro2):
    return parametro1 + parametro2, parametro1 ** parametro2;

resultado1 = suma(2,3);
print resultado1;
resultado2 = potencia(2,3);
print resultado2;
resultado3 = sumaPotencia(2,3);
print resultado3;
```

5. Tipos de datos en Python

En el apartado anterior hemos visto los tipos elementales de datos en Python: los enteros, los valores flotantes, las cadenas de texto y los valores booleanos. En este apartado, veremos tipos de datos más complejos que permiten trabajar con agrupaciones de estos tipos de datos básicos. En especial, veremos las tuplas, las listas, los conjuntos, los diccionarios y finalmente, los ficheros.

5.1. Tuplas

Una tupla es una secuencia inmutable y ordenada de elementos. Cada uno de los elementos que conforman una tupla puede ser de cualquier tipo (básico o no). La sintaxis para declarar una tupla consiste en especificar sus elementos separados por una coma (,).

Es muy habitual agrupar todos los elementos de una tupla entre paréntesis, aunque no es imprescindible (sólo se exige el uso del paréntesis cuando puede existir confusión con otros operadores). A menudo se coloca una coma al final de la tupla para indicar la posición del último elemento. En el código 5.1 se pueden ver distintos ejemplos de uso de tuplas.

Código 5.1: Ejemplo de tuplas en Python

```
# Ejemplos de uso de tuplas

tupla_vacia = ();
tupla1 = 1,2,3,4,6,
tupla1b = (1,2,3,4,6);
tupla2 = 'hola',2,3
tupla3 = tupla2,90
tupla4 = tuple('ejemplo');

#Resultado de la ejecución
>> print tupla1
(1, 2, 3, 4, 6)
>>> print tupla1b
(1, 2, 3, 4, 6)
>>> print tupla2
('hola', 2, 3)
>>> print tupla2
('hola', 2, 3)
>>> print tupla3
(('hola', 2, 3), 90)
>>> print tupla4
('e', 'j', 'e', 'm', 'p', 'l', 'o')
```

5.2. Listas

Una lista es una secuencia mutable y ordenada de elementos. A diferencia de las tuplas, los elementos de las listas se pueden modificar una vez han recibido un valor asignado. Para especificar los elementos que forman una lista, se usa una tira de elementos separada de nuevo por coma (,), al principio y al final la lista se envuelve entre corchetes [,].

En el código 5.2 se pueden ver ejemplos de usos de listas.

Código 5.2: Ejemplo de listas en Python

```
# Ejemplos de uso de listas

lista_vacia = [];
lista = [1,2,3,4,6]
lista2 = ['hola',2,3];
lista3 = [lista2,90,'good'];
lista4 = list('ejemplo');
lista5 = [1,2,3,4,5,6,7,8,9];

#Resultado de la ejecución

>>> print lista_vacia
[]
>>> print lista
[1, 2, 3, 4, 6]
>>> print lista2
['hola', 2, 3]
>>> print lista3
[['hola', 2, 3], 90, 'good']
>>> print lista4
['e', 'j', 'e', 'm', 'p', 'l', 'o']
>>> lista3[2]
'good'
>>> lista3[0]
['hola', 2, 3]

#Ejemplos de acceso a sublistas
>>> lista5[2:3]
[3]
>>> lista5[3:]
[4, 5, 6, 7, 8, 9]
>>> lista5[:3]
[1, 2, 3]
>>> lista5[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

#Añadir elementos a una lista
>>> lista = [1,2,3,4,6]
>>> lista.append([7,8,9])
>>> print lista
[1, 2, 3, 4, 6, [7, 8, 9]]
>>> lista = [1,2,3,4,6]
>>> lista.extend([7,8,9])
>>> print lista
[1, 2, 3, 4, 6, 7, 8, 9]

#Buscar y eliminar elementos
>>> lista2.index('hola')
0
>>> print lista2
['hola', 2, 3]
>>> lista2.remove('hola')
```

```
>>> print lista2
[2, 3]

#operadores sobrecargados de concatenación
>>> print lista2+lista2
[2, 3, 2, 3]
>>> print lista2*4
[2, 3, 2, 3, 2, 3, 2, 3]
>>>
```

Notad que los accesos a las listas (y también a las tuplas) se hacen mediante indexación directa. Para leer el primer elemento de una lista se usa la sintaxis **nombre_lista[0]**. Es importante destacar que los índices empiezan a contar desde 0, y llegan hasta el (número de elementos -1). Así, en el ejemplo, *lista3* tiene 3 elementos, el primero de los cuales (índice 0) es otra lista, el segundo (índice 1) es un 90 y el tercero (indice 2) 'good'. Existe también la posibilidad de referenciar los elementos de una lista con índices negativos. En este caso se empieza a contar desde el final de la lista, siendo el índice -1 el último elemento de la lista. Por ejemplo, lista3[-1] nos daría la palabra *good*, lista3[-2] un 90, y lista3[-3] la lista equivalente a la lista2.

Aparte de poder consultar los elementos individuales de las listas, también se puede acceder a una sublista. Para ello se usa el operador **:**. A la izquierda del operador se coloca el índice inicial y a la derecha el índice final. Si no se especifica ningún índice significa que se cogerán todos los elementos hasta llegar al extremo correspondiente. En el ejemplo, vemos cómo lista5[:3] nos da todos los elementos hasta el segundo (incluido), y lista5[3:] empieza por el tercero hasta el final de la lista. En el caso de lista5[:] nos devolverá toda la lista.

Para añadir elementos a una lista, disponemos de dos métodos, **append** y **extend**. Append añade un elemento a una lista (independientemente de su tipo), y extend recibe una lista de elementos, y los concatena a la lista actual. En el ejemplo se puede observar la diferencia de comportamiento de ambos métodos ante la misma entrada.

Ved también

Más adelante, en el apartado 6, dedicado a la orientación a objetos, se describirá con más detalle el concepto de método.

Otros métodos auxiliares útiles son la búsqueda de elementos y la eliminación de elementos. La búsqueda se suele realizar con el método **index**, que devuelve la posición donde se encuentra la primera ocurrencia del elemento. La eliminación se realiza con el método **remove** que elimina la primera ocurrencia de un determinado elemento en una lista. El ejemplo muestra el uso de estos dos métodos, donde se busca un determinado elemento.

Otra función muy útil en las listas, sobre todo para usar en bucles for, es **range**. Esta función, dado un número natural N, nos devuelve una lista con todos los elementos hasta N-1. Así pues, si tecleamos *range(8)* en la línea de comandos, obtendremos *[0, 1, 2, 3, 4, 5, 6, 7]*, que podrán ser seguidos en un bucle for para ejecutar una determinada acción 8 veces.

Finalmente, existen sobrecargas de los operadores aritméticos básicos +,*. El operador suma concatena dos listas, y el operador multiplicación recibe una lista y un valor escalar, devolviendo la repetición de la lista (tantas veces como indica el valor).

5.3. Diccionarios

Los diccionarios conforman uno de los tipos de datos más útiles para los programadores en Python, y representan una interesante novedad respecto a los lenguajes de programación imperativos tradicionales. Se trata de unas estructuras de memoria **asociativa**. Los elementos en los diccionarios no se consultan mediante un índice, sino que se accede directamente a ellos por contenido. El código 5.3 muestra un ejemplo de diccionario y su uso.

Código 5.3: Ejemplo de tuplas en Python

```
# Ejemplos de uso de diccionarios

diccionario1 = {'nombre': 'Alejandro', 'edad': 34, 'numerosfavoritos': (3,7,13)}

tupla = (('nombre', 'Alejandro'), ('edad', 34), ('numerosfavoritos', (3,7,13)))

diccionario2 = dict(tupla);
diccionario3 = {'libros':'el␣juego␣de␣Ender', 'telefono':934572345}

a = diccionario1['nombre']
>>> print a
Alejandro

>>> 'nombre' in diccionario2
True

diccionario1.update(diccionario3)
>>> print diccionario1
{'edad': 34, 'nombre': 'Alejandro', 'numerosfavoritos': (3, 7, 13), 'libros': 'el␣juego␣de␣Ender',
 'telefono': 934572345}

del diccionario1['nombre']
>>> print diccionario1
{'edad': 34, 'numerosfavoritos': (3, 7, 13), 'libros': 'el␣juego␣de␣Ender', 'telefono': 934572345}

>>> 'nombre' in diccionario1
False
```

Para crear el diccionario simplemente indicaremos entre llaves {,} las parejas de elementos que lo conforman. En cada pareja indicaremos primero el valor 'llave' para acceder al elemento, y después el valor que contendrá (que puede ser de cualquiera de los tipos disponibles en Python). La consulta de un valor del diccionario se reducirá a preguntar por el valor que tiene la llave asociada. En el ejemplo, guardamos en la variable **a** el contenido de la posición de memoria *nombre*. El string 'nombre' nos sirve para indexar posiciones de memoria o variables. Las estructuras de memoria asociativa que permiten esta funcionalidad son de gran utilidad en el tratamiento de datos textuales y los sistemas de clasificación de lenguaje natural.

Existen varios métodos auxiliares para trabajar con diccionarios, entre los que destacan el método **in**, que permite consultar si una determinada llave está presente en el diccionario, el método **update** que permite combinar dos diccionarios y el método **del** que elimina una entrada de un diccionario.

5.4. Conjuntos

Los conjuntos son secuencias ordenadas de elementos únicos (concepto matemático de conjunto). Existen dos tipos de conjuntos, los **sets** y los **frozensets** (que son mutables e inmutables respectivamente). Las operaciones típicas sobre conjuntos son: conocer la longitud de un conjunto (método **len**), conocer si un elemento está en un conjunto (método **in**), unión (método **union**), la intersección (método **intersection**), añadir un elemento a un conjunto (método **add**) y eliminar un elemento de un conjunto (método **remove**). El código 5.4 muestra un ejemplo de uso de conjuntos en Python.

Código 5.4: Ejemplo de conjuntos en Python

```
# Ejemplos de uso de conjuntos

conjunto1 = set([1,2,3,4])
conjunto2 = set([3,4])
conjunto2.remove(3)
conjunto2.add(5)
print conjunto2


uni = conjunto1.union(conjunto2)
int = conjunto1.intersection(conjunto2)
>>> print uni
set([1, 2, 3, 4, 5])
>>> print int
set([4])
```

5.5. Ficheros

Los ficheros no son un tipo de datos en sí mismos, sino un objeto que permite interactuar con el sistema de entrada y salida para escribir datos en el disco.

Observación

Hemos preferido explicar la funcionalidad básica de los ficheros en estos apuntes, puesto que pueden ser necesarios durante todo el curso para cargar los datos necesarios para los algoritmos de teoría.

En este manual, aún no hemos tratado el concepto de orientación a objetos en Python. Sin embargo, será necesario introducir algunos métodos básicos para acceder al objeto que hace de interfaz con el disco. La función básica de acceso a un fichero se denomina **open**, que recibe como parámetro un nombre de fichero y un modo (lectura “r”, escritura “w”). El acceso a los datos se puede realizar mediante los métodos:

- **read**: Lee un determinado número de bytes del fichero (argumento del método).
- **seek**: Se encarga de posicionar el lector en el byte indicado en el argumento.

- **tell**: Devuelve la posición que se está leyendo en la actualidad del fichero.
- **close**: Cierra el fichero.

Código 5.5: Ejemplo de acceso a un fichero. Ejecutar el código y verificar el funcionamiento de las funciones básicas de acceso a fichero

```
# Ejemplos de uso de ficheros

f=open("files.py","r")
f.tell()

f.read(30)

f.read(10)
f.tell()

f.seek(2)
f.tell()
f.read(8)
f.close()
```

5.5.1. Ficheros de texto

Los ficheros de texto son un caso particular que Python permite tratar de forma mucho más cómoda e integrada en el lenguaje. Existe un método **readlines()**, que permite leer el contenido del fichero por líneas, devolviendo un array donde se puede indexar cada una de ellas. Del mismo modo, es posible acceder línea por línea al contenido de un fichero, y efectuar un tratamiento individualizado de cada una de ellas mediante un simple bucle for. En el ejemplo del código 5.6 podéis ver el código que recorre un fichero y lo imprime por pantalla.

Código 5.6: Ejemplo de acceso a un fichero

```
# Ejemplos de uso de ficheros

#Leer una linea
f = open("Textfiles.py", "r")
lineas = f.readlines()

#imprimir por pantalla la primera linea
print lineas[0]
#imprimirlas todas
print lineas[:]

#Cerrar el fichero
f.close();

#Leer todo el fichero linea a linea
for line in open("Textfiles.py", "r"):
    print line

#Bucle que recorre las lineas de un fichero origen y las
#copia en el destino
fwrite = open("CopiaTextfiles.py", "w")
for line in open("Textfiles.py", "r"):
    fwrite.write(line);

fwrite.close()
```

Finalmente, disponemos del método **write** que permite escribir contenidos en un fichero. En el ejemplo anterior, recorremos el primer fichero y línea a línea lo vamos escribiendo en el fichero destino ("CopiaTextfiles").

6. Python y la orientación a objetos

El lenguaje Python permite trabajar mediante el paradigma de programación imperativa clásica (como en todos los ejemplos vistos hasta ahora), o mediante el paradigma de la orientación a objetos, e incluso mediante lenguaje funcional. En este apartado veremos la sintaxis de la orientación a objetos (OO) en Python, con algunos ejemplos prácticos. Asumiremos que el estudiante conoce las bases de la OO, y que está familiarizado con los conceptos de clase, objeto y método.

6.1. Los objetos en Python

Un objeto no es más que el resultado de encapsular una determinada entidad, que está formada por un estado (los datos o **atributos**) y un funcionamiento (los **métodos**). Una clase es la plantilla o concepto genérico de un objeto, que se usa para definir sus propiedades y servicios. Un objeto es pues una instancia concreta de una clase.

Para definir una clase usamos la palabra clave **class**. En el fragmento de código 6.1 mostramos un ejemplo de definición de una clase.

Código 6.1: Ejemplo de definición de una clase

```
# Definición de la clase Panaderia
class Panaderia:
    def __init__(self, panes, pastitas):
        self.panes = panes
        self.pastas = pastitas
        print "En la tienda hay", self.panes, "panes y",pastas, "pastas aunque estas no se venden"
    def vender(self):
        if self.panes > 0:
            print "Vendido un pan!"
            self.panes -= 1
        else:
            print "Lo sentimos no quedan panes por vender"
    def cocer(self, piezas):
        self.panes += piezas
        print "Quedan", self.panes, "panes"

panaderia1 = Panaderia(3,4)
panaderia2 = Panaderia(1,2)

panaderia2.vender()
panaderia2.vender()

panaderia2.cocer(1)
panaderia2.vender()
```

Lo primero que observamos, después de los comentarios, es el uso de la palabra clave **class**, que nos permite definir el entorno de una clase. A continuación viene el nombre de la clase, *Panadería*, que se utilizará posteriormente para construir objetos de la clase. Para definir los métodos de la clase, hacemos uso de nuevo de la palabra clave def. Los métodos se definen siguiendo una sintaxis bastante parecida a las funciones.

Ved también

La palabra clave **def** se estudia en el apartado 4, dedicado a las funciones.

Otro punto destacable es el método **__init__**. Este método es especial, siempre recibe el mismo nombre (independientemente de la clase), y sirve para especificar las acciones a llevar a cabo en el momento de la creación e inicialización de un objeto de la clase en cuestión. En este caso, recibe tres parámetros, aunque el primero de ellos es de nuevo especial, se trata de la palabra reservada **self**. **Self** se utiliza para hacer referencia al propio objeto. Una vez creado, **self** nos permitirá diferenciar los nombres de las variables miembro del objeto del resto de valores. La función del **self** es parecida a **this** en otros lenguajes de programación (C++, JAVA,...). Si seguimos mirando el ejemplo, observamos cómo tenemos una variable miembro llamada *panes*, y un argumento a la función **__init__** con el mismo nombre. En la expresión (**self.panes=panes**) diferenciamos ambos valores mediante el uso del **self**.

6.1.1. Creación de instancias (objetos) de una clase

Para crear un objeto concreto de una clase, basta con usar el nombre que le hemos dado a la clase seguido de los argumentos que necesita la función de inicialización (que aquí tiene un efecto similar a los constructores del lenguaje C++). En el ejemplo hemos creado dos objetos de la clase Panadería (*panaderia1* y *panaderia2)*.

La ejecución de métodos, como en muchos lenguajes de programación, sigue la sintaxis **nombre_del_objeto.nombre_del_metodo(argumentos,...)**.

En el ejemplo se llama en diversas ocasiones a los métodos *vender* y *cocer* de los objetos creados. Ejecutad el fragmento de código e interpretar el resultado que se imprime por pantalla.

6.2. Herencia

Cuando decimos que una clase hereda de otra, nos referimos a que la clase resultante contendrá todos los atributos y métodos de la clase *padre* o *superclase*, permitiendo de esta forma una especialización progresiva de las clases y una mayor reutilización de código.

Para hacer que una clase herede de otra en Python, simplemente tenemos que indicarlo a continuación del nombre de la clase, entre paréntesis. En el código 6.2 se muestra un ejemplo muy simple de herencia.

Código 6.2: Ejemplo de uso de herencia en las clases Python

```
# Definición de la clase Animal: Ejemplos de Herencia
class Animal:
        def __init__(self, age, weight):
                self.age = age
                self.__weight = weight
        def __privateMethod(self):
                print self.weight;
        def getWeight(self):
                return self.__weight;
        def eat(self, kgm):
                self.__weight += kgm;
                print "The animal Weitghs:", self.__weight,"after eating."

class Bird(Animal):
        def fly(self):
                print "I fly as a bird!"

class Mammal(Animal):
        def fly(self):
                print "I can not fly, I am a mammal!"

class Ostrich(Animal,Bird):    #Avestruz
        def fly(self):
                print "I can not fly, I am a Bird but ostrichs do not fly!"

class Platypus1(Mammal, Bird):
        pass

class Platypus2(Bird,Mammal):
        pass


animal1 = Animal(3,0.5)
animal1.eat(0.1)

canary = Bird(1,0.45)
canary.eat(0.02)
canary.fly()

bear = Mammal(10,150)
bear.eat(10)
bear.fly()

ostrich = Ostrich(5,30)
ostrich.fly()

platypus = Platypus1(2,3)
platypus.fly()

platypus = Platypus2(2,3)
platypus.fly()

print bear.getWeight()
bear.privateMethod()
```

Observad que en este ejemplo trabajamos con cuatro clases distintas: *Animal*, *Bird*, *Mammal* y *Ostrich*. Animal es superclase del resto, que heredan de ella. En el caso de *Ostrich*, notad que hereda de dos clases a la vez (aunque no fuese estrictamente necesario). Python permite herencia múltiple. Ejecutad el

código, y entender su funcionamiento. Observad que el método *fly()* tiene un comportamiento diferente en función del tipo concreto del objeto.

Muchos lenguajes no permiten herencia múltiple, dado que esto puede originar conflictos cuando se heredan métodos o atributos con el mismo nombre de dos superclases.

En caso de conflicto, Python da preferencia a la clase situada más a la izquierda en el momento de su definición.

Volver a ejecutar el ejemplo, y observad el resultado de las llamadas a *fly* de las dos clases *Platypus1* y *Platypus2*. Este ejemplo sintético ilustra el orden de preferencias en función del orden de definición de la herencia en caso de conflictos de nombre.

6.3. Encapsulación

Una de las principales ventajas de la programación orientada a objetos es la encapsulación. Esta propiedad permite construir objetos con métodos y atributos que no pueden llamarse externamente. Se trata de código interno que el programador ha preparado y que no quiere que se vea alterado. El objeto ofrece una serie de servicios al exterior, ocultando parte de su codificación interna. En Python no tenemos palabras clave específicas para denominar la encapsulación. Todos los métodos en Python son públicos, excepto aquellos que empiezan por un doble guión bajo (__). En el ejemplo anterior, la llamada a *bear.privateMethod()* produce una excepción en tiempo de ejecución.

Encapsulación en otros lenguajes

En otros lenguajes sí se utilizan palabras clave para denominar la encapsulación. Por ejemplo,en C++ o JAVA, que usan *public, private, protected*,...

La encapsulación tiene mucha utilidad si se quieren esconder los detalles de implementación de una determinada clase. Supongamos por ejemplo que internamente el peso de una clase Animal se guardase en otras unidades (sistema inglés) en lugar de kilogramos. El uso de funciones setters y getters públicas permitiría que la interfaz con el usuario fuese siempre la misma, independientemente de esta codificación interna, que sería privada. Se ha ejemplificado este hecho en el método **getWeight()** de la clase *Mammal*.

Ved también

El concepto de getter y setter se ha tratado con profundidad en la asignatura *Programación orientada a objetos*.

Como curiosidad, observad que la función **__init__** empieza también por dos guiones bajos. Los métodos Python que empiezan y terminan por guiones bajos son especiales, y no tienen que ver con el concepto de método privado. Otra función especial es el destructor o **__del__** que se llama cuando el objeto deja de utilizarse para su eliminación.

6.4. Polimorfismo

En el caso de Python, como en la mayoría de lenguajes de programación, se basa en el uso de herencia.

La palabra polimorfismo* en programación denomina a la propiedad que tienen muchos lenguajes de ejecutar código distinto en función del objeto que hace la llamada.

*** Del griego *varias formas*.**

Así, podríamos referenciar objetos mediante el tipo superclase, pero que en el momento de ejecutar sus métodos, se llamará realmente a los métodos de la clase derivada. El concepto de polimorfismo se encuentra muy ligado al enlace dinámico. Python por defecto ya usa el enlace dinámico, con lo que no requiere ninguna notación especial para usar polimorfismo.

Enlace dinámico

El enlace dinámico es la decisión de qué código se ejecuta en tiempo de ejecución en lugar de en tiempo de compilación.

7. Python como lenguaje funcional

La programación funcional es un paradigma de programación basado en el concepto matemático de función, no en un sentido procedural como hemos visto hasta ahora, sino más bien en el uso de funciones de orden superior. Este concepto hace referencia al uso de las funciones como si fueran valores propios del lenguaje.

> Es decir, en la programación funcional podemos guardar una función en una variable, para posteriormente aplicarla sobre unos argumentos, permitiendo incluso que una función retorne otra función como salida.

Las características funcionales de Python no van a ser las más utilizadas en este curso, pero en este apartado haremos un breve resumen con algún ejemplo orientativo, puesto que existen algunos iteradores bastante utilizados que se basan en el paradigma funcional.

El código 7.1 es un ejemplo en el que se muestra el uso básico de la programación funcional, el acceso a las funciones como variables.

Código 7.1: Ejemplo simple de aplicación de Python funcional

```
# Ejemplo de uso de lenguaje funcional

def money(country):
    def spain():
        print "Euro"
    def japan():
        print "Yen"
    def eeuu():
        print "dollar"
    functor_money ={"es":spain,
                    "jp":japan,
                    "us":eeuu}
    return functor_money[country]

f = money("us")
money("us")()
f()

f = money("jp")
f()
```

Introducid el ejemplo y ejecutadlo. Como podréis observar, la línea de código *f = money("us")* no realiza ninguna acción visible. Esta línea se encarga de crear una nueva variable, que será una función. Esta función se genera o selecciona en función de la entrada del usuario, en este caso, una cadena de

caracteres que activa el selector en el diccionario interno a la función money. El resultado guardado en *f* es una función, y por tanto puede ser llamado para su ejecución. Como podéis observar, las llamadas *money("us")()* y *f()* son equivalentes. A posteriori hemos cambiado el valor donde "apunta" la función *f*, modificando en consecuencia su codificación (y por tanto, generando una salida distinta).

El paradigma funcional tiene múltiples ventajas y aplicaciones. En este manual nos centraremos en describir tres iteradores que se han usado mucho en conjunción con las listas: **map**, **filter** y **reduce**. El estudiante familiarizado con los lenguajes LISP o ML verá cierto paralelismo con estos iteradores. El código 7.2 muestra su ejemplo de uso.

Código 7.2: Ejemplo de aplicación de iteradores

```
# Ejemplo de uso de iteradores

def double(num):
    return num*2
def even(num):
    return (num%2) == 0
def operation(num1,num2):
    return num1*num2+1;

l1 = range(10);

l2 = map(double,l1);
l3 = filter(even,l1);
l4 = reduce(operation,l1);

#Equivalente pero usando lambda function
l5 = map(lambda num: num*2,l1)
l6 = filter(lambda num: num%2==0,l1)
l7 = reduce(lambda num1,num2: num1*num2 +1,l1)

#Comprensión de listas
l8 = [num*2 for num in l1];

print l1
print l2,l5
print l3,l6
print l4,l7

print l8
```

Observad que la sintaxis de **map**, **filter** y **reduce** es similar. Siempre reciben un primer argumento, que es el nombre de la función que van a ejecutar sobre los elementos de la lista (pasada como segundo argumento). En el caso de **map**, el iterador aplica la función a cada elemento de la lista, devolviendo una nueva lista con el resultado de aplicar esta función a cada elemento. Por su parte, **filter**, devuelve una lista con aquellos elementos de la lista original que pasan la evaluación de la función. En este caso la función que recibe **filter** devuelve un valor booleano (*true, false*) que hace las funciones de selección. Finalmente **reduce** aplica recursivamente la función a cada par de elementos de la lista, hasta dejar un solo resultado.

7.1. Funciones lambda

Las funciones lambda son funciones anónimas definidas en línea, que no serán referenciadas posteriormente. Se construyen mediante el operador **lambda**, sin usar el paréntesis para indicar los argumentos. Estos van directamente después del nombre de la función, que finaliza su declaración con dos puntos (**:**). Justo después, en la misma línea se escribe el código de la función. En el ejemplo anterior se ha incluido la versión de llamada a **map**, **filter** y **reduce** usando funciones lambda. Observad que estas funciones están limitadas a una sola expresión.

7.2. Comprensión de listas

Una sintaxis alternativa a los iteradores anteriores que se está imponiendo en las últimas versiones de Python es la comprensión de listas. En este caso, se pretende crear una lista a partir de otra lista. La sintaxis es de nuevo muy sencilla, se especifica entre corchetes (o paréntesis) la expresión a aplicar, seguida de la palabra clave **for**, la variable a iterar, la palabra clave **in** y la lista origen. Como se puede observar en el ejemplo anterior, hemos reproducido la función que duplicaba cada elemento de la lista aplicando comprensión, y la expresión se leería: "para cada num de l1 haz num*2".

Pattern matching

A diferencia de otros lenguajes puramente funcionales, Python no implementa *pattern matching*, aunque es posible simularlo. Estos conceptos quedan ya fuera del alcance de este manual.

8. Librerías: NumPy PyLab y SciPy

El código Python se puede agrupar en módulos y paquetes para mejorar su organización y poder reutilizar y compartir todo lo programado. Cada fichero equivale a un módulo. Para poder utilizar código de estos ficheros se utiliza la palabra clave **import**, seguida del nombre del módulo, del mismo modo que en C usábamos **include**. Hay que tener en cuenta que **import** carga literalmente todo el contenido del fichero, ejecutando también toda la parte ejecutable dentro del módulo. Es habitual definir módulos donde sólo se definan las funciones y clases que se quieren publicar para evitar ejecuciones involuntarias. También es posible importar sólo un objeto concreto que nos interese, esto se hace mediante la construcción **from** módulo **import** nombre del objeto. Las llamadas a objetos suelen ir precedidas con el nombre del módulo donde se encuentran, para preservar el espacio de nombres y mantener objetos con el mismo nombre en distintos módulos.

Python tiene asociada una variable de entorno (PYTHONPATH) en la que se le indica dónde encontrar (la carpeta concreta) la mayor parte de las librerías integradas en el lenguaje. En caso de crear nuevos módulos en otra ubicación distinta del código actual, se puede añadir esta ruta a la variable de entorno PYTHONPATH.

Los paquetes no son más que entidades que organizan los módulos. Los paquetes tienen su equivalencia con el sistema de carpetas o directorios, donde podemos guardar de forma estructurada diferentes ficheros (módulos). La palabra clave **import** se utiliza también para importar módulos de los paquetes, y se usa la misma nomenclatura en cuanto a espacio de nombres. Así pues, para importar un módulo llamado *moduloEjemplo*, del paquete *PaquetesEjemplo*, seguiríamos la sintaxis: **import** *PaquetesEjemplo.moduloEjemplo*. Suponiendo que en este módulo tuviéramos de nuevo definida la función double, la llamaríamos de la siguiente manera: *PaquetesEjemplo.moduloEjemplo.double(3)*, acción que nos devolvería un 6. Esta notación se puede simplificar notablemente con la palabra clave **as**. Siguiendo el ejemplo, si importáramos el código como: **import** *PaquetesEjemplo.moduloEjemplo* **as** *e*, podríamos llamar posteriormente a la función double como *e.double(3)*.

En este curso trabajaremos con varios paquetes de Python, pero tres de ellos serán imprescindibles puesto que contienen las principales librerías de *machine learning* ya implementadas en Python. Se trata de NumPy, PyLab y SciPy.

8.1. NumPy

La librería NumPy nos permite trabajar con datos científicos, equiparando en cierta forma el potencial de Python al de otros lenguajes como Matlab o Octave. NumPy se encuentra disponible de forma gratuita en Internet*. Desde allí se puede bajar y consultar la documentación de las diferentes propiedades que nos ofrece NumPy, que se centran básicamente en el tratamiento de matrices y arrays N-dimensionales, y en un conjunto de funcionalidades de álgebra lineal y tratamiento de la señal aplicada al análisis científico.

*** http://numpy.scipy.org/**

8.2. PyLab

PyLab es una librería que intenta aportar funcionalidades extra a NumPy integrando gran parte de las funciones Matlab que se han usado históricamente en entornos de *machine learning*. En Internet* se puede encontrar el paquete listo para descargar, con la correspondiente documentación.

*** http://www.scipy.org/PyLab**

8.3. SciPy

SciPy es una expansión de NumPy, que integra nuevos paquetes para el tratamiento científico de datos. Integra gran cantidad de funciones de procesamiento de imágenes, procesado de la señal, estadística, integración numérica. En Internet* se puede encontrar la última versión de la plataforma y su documentación asociada.

*** http://www.scipy.org/**

Resumen

En este módulo hemos visto los elementos básicos de la programación en Python. El módulo está pensado para que un estudiante con conocimientos de programación pueda rápidamente introducirse en Python, y pueda realizar sus primeros programas en poco tiempo. El objetivo primordial del módulo y de la asignatura es el correcto seguimiento de los conceptos expuestos en teoría. No se pretende formar expertos programadores en Python, sino más bien capacitar al estudiante para poder entender el abundante código de ejemplo que incorporan los materiales.

Es aconsejable que realicéis los ejercicios para acabar de asentar los conocimientos básicos. No es necesario realizarlos todos, simplemente intentar hacer aquellos que a primera vista parezcan más complejos. Junto con el material, podréis encontrar en el aula las soluciones a los ejercicios, para poder realizar las consultas y comparaciones oportunas.

Ejercicios de autoevaluación

1. Escribid una función en Python que, dada una lista de números, devuelva otra lista en orden inverso. Para realizar este ejercicio se deberá utilizar un bucle o estructura repetitiva. No se permite el uso de funciones miembro de la clase list (en especial list.reverse()).

2. Escribid una función que, dado un número entero N, devuelva una lista con todos los números primos hasta N. Para solucionar el ejercicio debéis crear una función auxiliar que indique si un determinado número es primo (retornando un valor booleano).

3. Escribid una función que reciba una tupla compuesta por caracteres, y devuelva una lista con los caracteres en mayúsculas. Debéis recorrer la tupla carácter a carácter para realizar la conversión. Para convertir un carácter a mayúscula podéis usar el método upper(). Por ejemplo 'a'.upper() nos devuelve 'A'.

4. Convertid el texto 'ejemplo' en una lista que contenga sus 7 caracteres. Después convertidlo en una tupla y usando la función del ejercicio anterior obtened una lista con el texto en mayúsculas.

5. Escribid una función que, dada una lista de números, devuelva una lista con sólo los elementos en posición par.

6. Extended la función anterior para que, dada una lista y unos índices, nos devuelva la lista resultado de coger sólo los elementos indicados por los índices. Por ejemplo si tenemos la lista [1,2,3,4,5,6] y los índices [0,1,3] debería devolver la lista [1,2,4].

7. Escribid una función que nos devuelva cuántas veces aparece cada una de las palabras de un texto (frecuencia de aparición de las palabras). Para ello podéis usar un diccionario donde la llave sea cada una de las palabras del texto y el contenido guarde el número de apariciones de la palabra. Para simplificar el ejercicio, podéis usar el método split(' '), que, dado un separador (el espacio), nos devuelve una lista con todas las palabras de un texto de forma separada. Por ejemplo: 'hola esto es un ejemplo'.split(' ') nos devolvería: ['hola', 'esto', 'es', 'un', 'ejemplo']

8. Escribid una función que devuelva un conjunto formado por los números compuestos (no primos) menores que un N dado.

9. Codificad una función que escriba en un fichero de texto los números primos que van desde el 1 hasta el 999.999.

10. Escribid una función que lea el contenido de un fichero de texto y nos dé la frecuencia de aparición de cada palabra. Podéis usar el código del ejercicio 7, en el que se usaban diccionarios para contar las apariciones de cada palabra.

11. Implementad un programa que tenga dos clases, Camión y Coche, ambas subclases de la superclase Vehículo. Elegid tres atributos comunes a Coche y Camión y dos atributos específicos a cada clase. Pensad bien dónde colocar cada atributo. Escribid un mínimo de dos métodos en cada clase y ejecutadlos en el programa principal.

12. Escribid una versión del ejercicio 2 que utilice programación funcional. Podéis usar el iterador filter para mantener sólo aquellos valores de la lista que sean primos.

13. Escribid una función que dependiendo de un selector ejecute alguno de los primeros 5 ejercicios de este apartado. La función recibirá un carácter ('1','2',...,'5') y deberá devolver una función que testea el apartado correspondiente. Por ejemplo, si escribimos f=selector('4'), f deberá ser una función que al ejecutarse finalmente nos devuelva la palabra *ejemplo* en mayúsculas.

Bibliografía

González Duque, Raúl. *Python para todos*. CC. (Version on-line)

Lutz, Mark (2011). *Programming Python* (4.ª ed.). O'Reilly Media.

Lutz, Mark (2007). *Learning Python* (3.ª ed.). O'Reilly Media.

Martelli, Alex (2006). *Python in a Nutshell* (2.ª ed.). O'Reilly Media.

Pilgrim, Mark (2004). *Dive Into Python*. APress (Versión on-line).